RBE500 - Foundations of Robotics

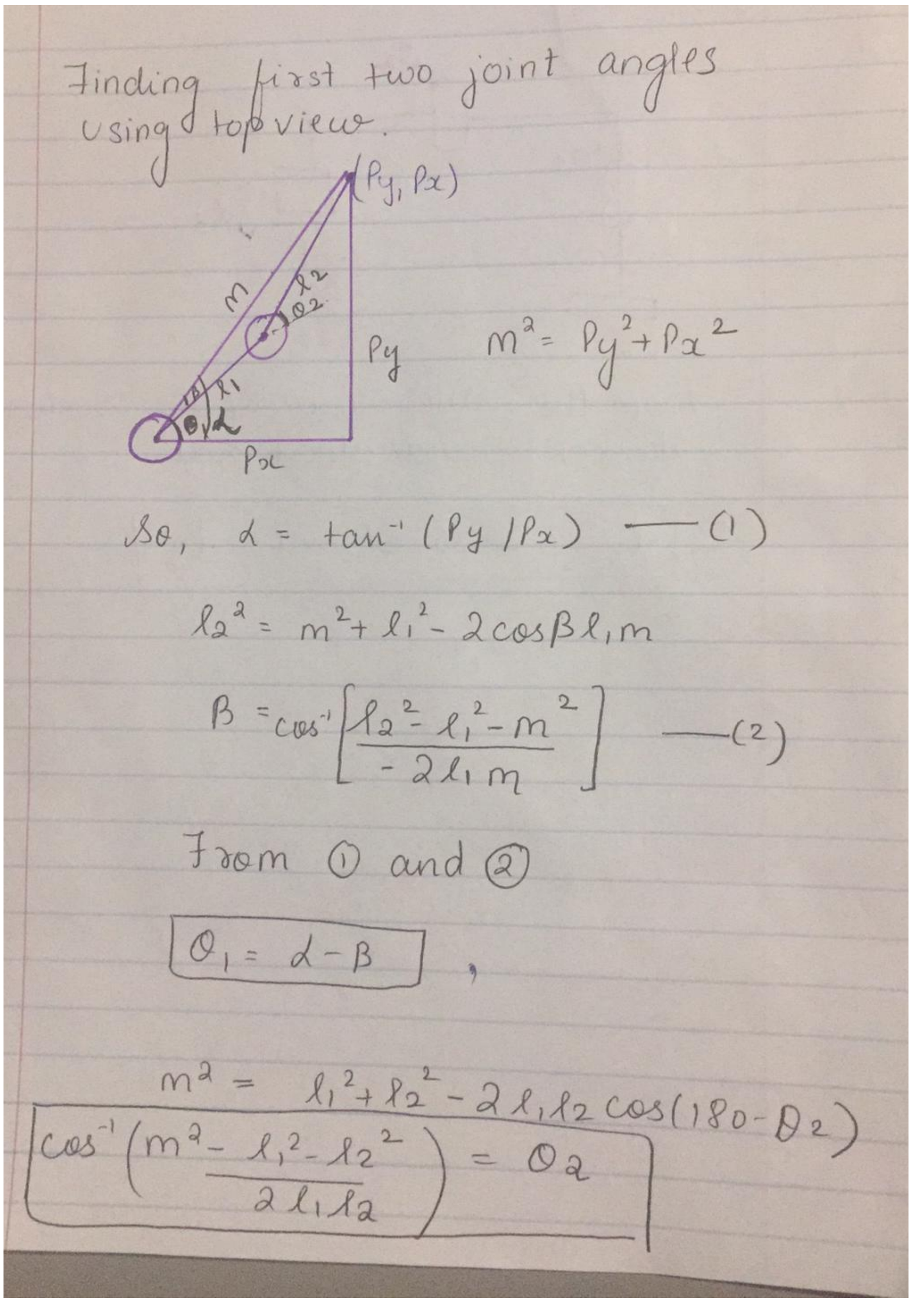
Final Group Project Report

Submitted by:
Suketu Parekh, Prarthana Sigedar

1. Objective

The objective of this project is to develop a joint space position control for a 3-DOF revolute-revolute-prismatic (RRP) robot manipulator. The task is to implement a PID controller for joint space tracking of the robot in Gazebo.

2. Inverse Kinematics



(Py, Px)
m
l2
θ2
l1
β
θ1
α
Py
Px

Finding first two joint angles using top view.

$$m^2 = P_y^2 + P_x^2$$

So, $\alpha = \tan^{-1}(P_y / P_x)$ —(1)

$$l_2^2 = m^2 + l_1^2 - 2\cos\beta\, l_1 m$$

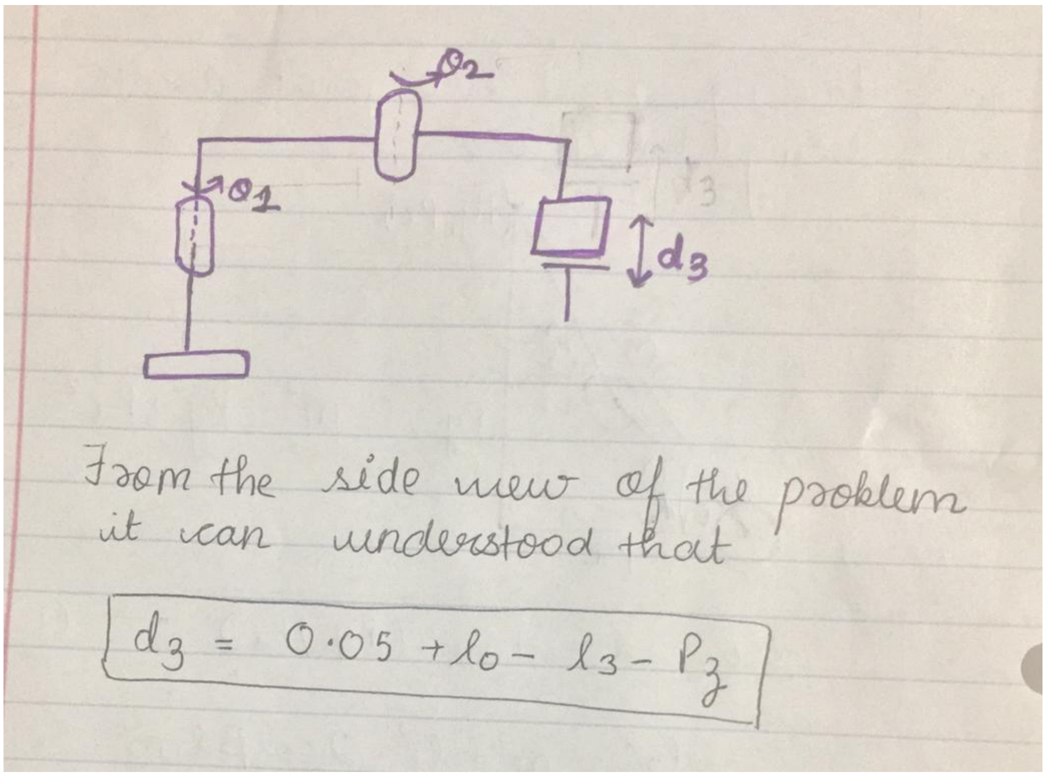
$$\beta = \cos^{-1}\left[\frac{l_2^2 - l_1^2 - m^2}{-2 l_1 m}\right] \quad \text{—(2)}$$

From ① and ②

$$\boxed{\theta_1 = \alpha - \beta} \; ,$$

$$m^2 = l_1^2 + l_2^2 - 2 l_1 l_2 \cos(180 - \theta_2)$$

$$\boxed{\cos^{-1}\left(\frac{m^2 - l_1^2 - l_2^2}{2 l_1 l_2}\right) = \theta_2}$$



From the side view of the problem it can understood that

$$\boxed{d_3 = 0.05 + l_0 - l_3 - P_z}$$

3. PID Controller Design

The PID controller was implemented in the run function inside the client code. The PID equation implements the controller in a discrete fashion. Initially, for the differential part of the PID equation, the difference between the consecutive errors was used along with a multiplication factor (of 100) to account for the frequency, as in the inverse of the time delay between which the errors are subtracted.

```
u1 = Kp_1*error1 + Kd_1*100*error_dot1 + Ki_1*E1
```

The above equation is the equation used initially for the control effort for Joint 1 of the RRP manipulator.

```
error_dot1 = error1 - error_prev1
```

is the difference between the consecutive errors.

Later, it was realized that a continuous approach for the differential part could yield better results, and hence the negative velocity was used in order to account for the time rate of change of error.

```
u1 = Kp_1*error1 - Kd_1*self.vel_j_1 + Ki_1*E1
```

The above equation represents the final equation used for calculating the joint effort for Joint 1. Here, `self.vel_j_1` represents the joint velocity for Joint 1.

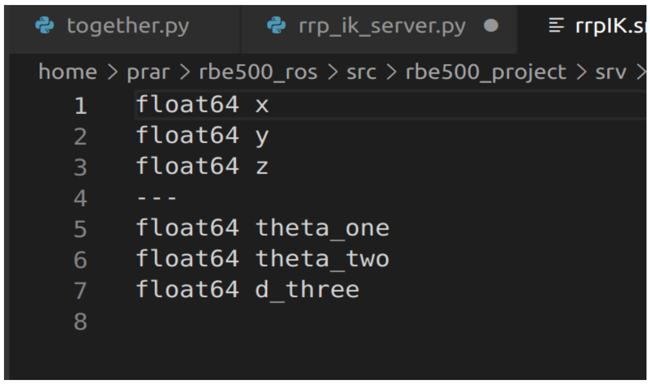
The coefficients (P, I, D gains) were tuned by trial and error to reduce and minimize the overshoot and oscillation problems.

4. Code Implementation

a. Part one

We have implemented inverse kinematics using a Service and Client.

Our service file is defined in srv folder and has two parts: Request and Response, which are defined as follows:



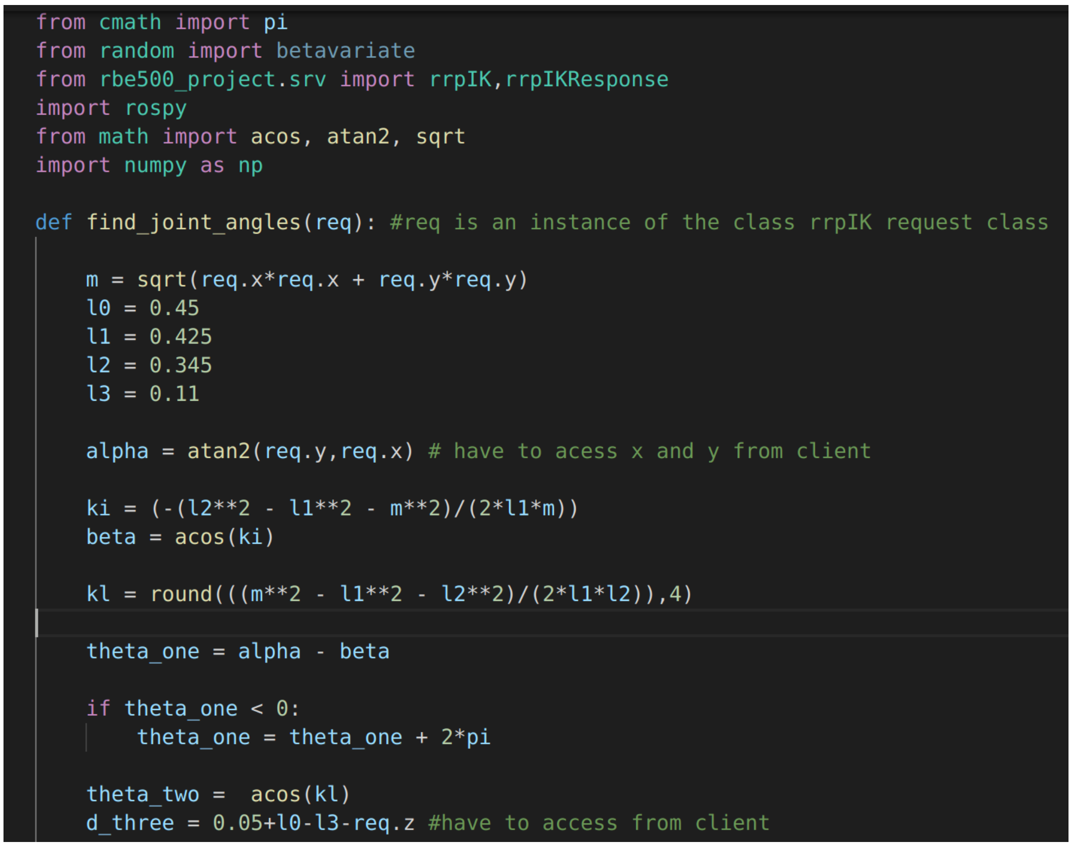
```
float64 x
float64 y
float64 z
---
float64 theta_one
float64 theta_two
float64 d_three
```

In the **request part,** we have three variables ie. x,y, and z which denote the position of the end effector.

In the **response part,** we have three outputs ie. theta_one, theta_two, and d_three which return the value of joint variables using inverse kinematics.

The client calls the server node by sending data (end-effector positions) and waits for the reply. So, the server node operates by calculating the inverse kinematics and reverts with the response to the client i.e theta_one, theta_two, and d_three in this case.

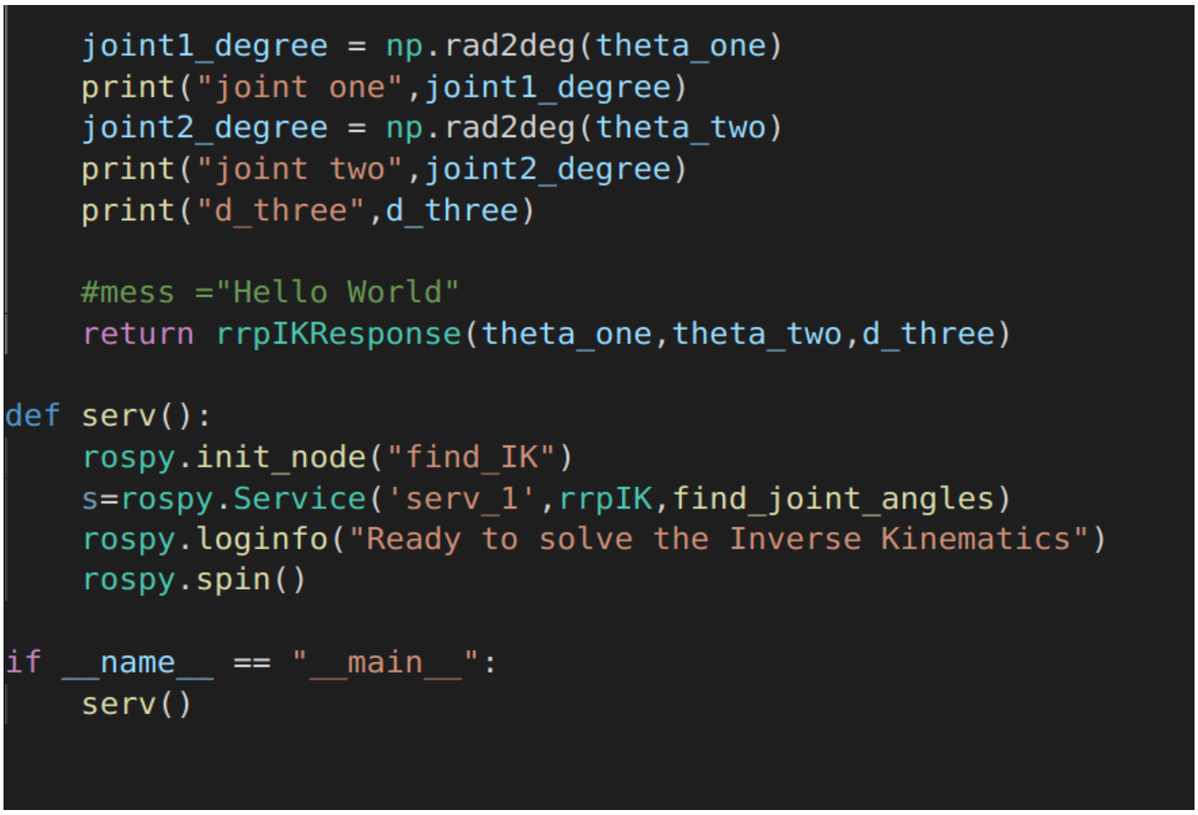
The code for calculating inverse kinematics is shown below:-



```
from cmath import pi
from random import betavariate
from rbe500_project.srv import rrpIK,rrpIKResponse
import rospy
from math import acos, atan2, sqrt
import numpy as np

def find_joint_angles(req): #req is an instance of the class rrpIK request class

    m = sqrt(req.x*req.x + req.y*req.y)
    l0 = 0.45
    l1 = 0.425
    l2 = 0.345
    l3 = 0.11

    alpha = atan2(req.y,req.x) # have to acess x and y from client

    ki = (-(l2**2 - l1**2 - m**2)/(2*l1*m))
    beta = acos(ki)

    kl = round(((m**2 - l1**2 - l2**2)/(2*l1*l2)),4)

    theta_one = alpha - beta

    if theta_one < 0:
        theta_one = theta_one + 2*pi

    theta_two =  acos(kl)
    d_three = 0.05+l0-l3-req.z #have to access from client
```

In the above figure, we have defined all the link lengths and calculated the inverse kinematics for each joint using the top view and side view. Moreover, it can be noticed that we have rounded our values of kl, which is used to find **theta_two**. The reason being, values of kl were coming around 1.00002 and when we do the inverse cosine of this value, we were getting a math domain error, hence we decided to round the values.

We also restricted our **theta_one** value, as for the third desired end-effector position we were getting -180 and hence we decided to add a condition that would restrict the joint one to be positive.



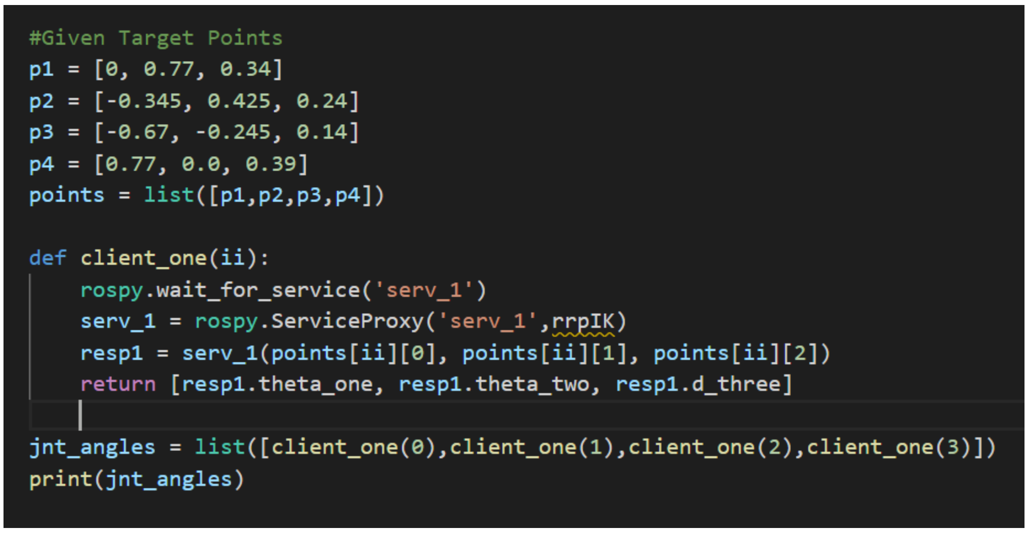
```
    joint1_degree = np.rad2deg(theta_one)
    print("joint one",joint1_degree)
    joint2_degree = np.rad2deg(theta_two)
    print("joint two",joint2_degree)
    print("d_three",d_three)

    #mess ="Hello World"
    return rrpIKResponse(theta_one,theta_two,d_three)

def serv():
    rospy.init_node("find_IK")
    s=rospy.Service('serv_1',rrpIK,find_joint_angles)
    rospy.loginfo("Ready to solve the Inverse Kinematics")
    rospy.spin()

if __name__ == "__main__":
    serv()
```

After calculating the angles, we have defined a function to initiate the node named “ **find_IK** ” followed by rospy.Service, which in turn starts the service and calculates the joint angles whenever requested by the client to do so.

b. Part two:

Part two consists of two subsections and code implementation is explained for the same in the following section:

a. The client node
b. Three PID controllers to control three joints

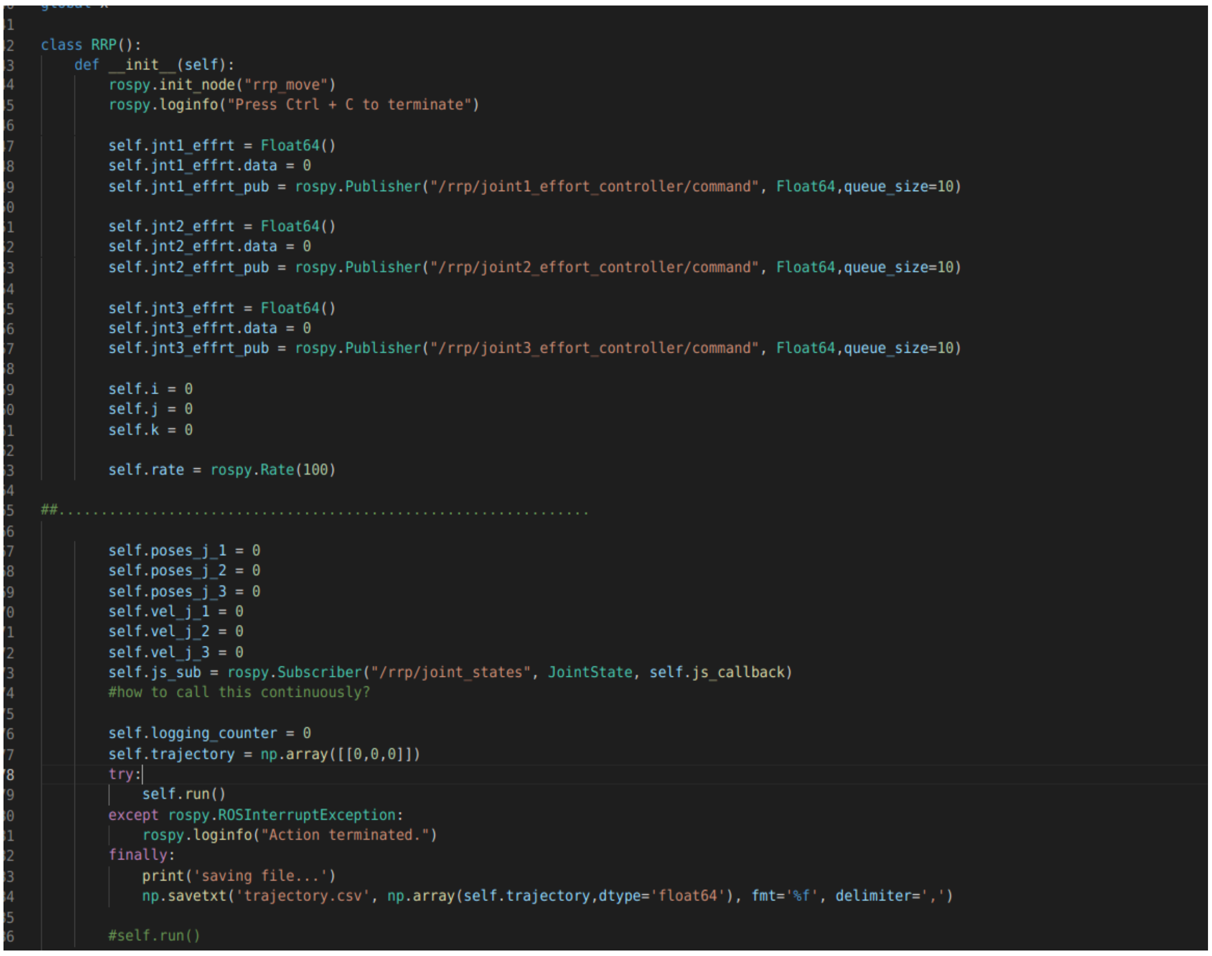


```
#Given Target Points
p1 = [0, 0.77, 0.34]
p2 = [-0.345, 0.425, 0.24]
p3 = [-0.67, -0.245, 0.14]
p4 = [0.77, 0.0, 0.39]
points = list([p1,p2,p3,p4])

def client_one(ii):
    rospy.wait_for_service('serv_1')
    serv_1 = rospy.ServiceProxy('serv_1',rrpIK)
    resp1 = serv_1(points[ii][0], points[ii][1], points[ii][2])
    return [resp1.theta_one, resp1.theta_two, resp1.d_three]

jnt_angles = list([client_one(0),client_one(1),client_one(2),client_one(3)])
print(jnt_angles)
```

Firstly, all the end effector positions are saved in the list called **points**.

We have defined a function named "**client_one**" which waits for the service to start. After that, we send end-effector positions to the server node in order to find the joint angles of all the desired positions and then save them in the list named "**jnt_angles**".

For part two, at first we defined class RRP in which we executed our PID controller.



```python
class RRP():
    def __init__(self):
        rospy.init_node("rrp_move")
        rospy.loginfo("Press Ctrl + C to terminate")

        self.jnt1_effrt = Float64()
        self.jnt1_effrt.data = 0
        self.jnt1_effrt_pub = rospy.Publisher("/rrp/joint1_effort_controller/command", Float64,queue_size=10)

        self.jnt2_effrt = Float64()
        self.jnt2_effrt.data = 0
        self.jnt2_effrt_pub = rospy.Publisher("/rrp/joint2_effort_controller/command", Float64,queue_size=10)

        self.jnt3_effrt = Float64()
        self.jnt3_effrt.data = 0
        self.jnt3_effrt_pub = rospy.Publisher("/rrp/joint3_effort_controller/command", Float64,queue_size=10)

        self.i = 0
        self.j = 0
        self.k = 0

        self.rate = rospy.Rate(100)

##..............................................................

        self.poses_j_1 = 0
        self.poses_j_2 = 0
        self.poses_j_3 = 0
        self.vel_j_1 = 0
        self.vel_j_2 = 0
        self.vel_j_3 = 0
        self.js_sub = rospy.Subscriber("/rrp/joint_states", JointState, self.js_callback)
        #how to call this continuously?

        self.logging_counter = 0
        self.trajectory = np.array([[0,0,0]])
        try:
            self.run()
        except rospy.ROSInterruptException:
            rospy.loginfo("Action terminated.")
        finally:
            print('saving file...')
            np.savetxt('trajectory.csv', np.array(self.trajectory,dtype='float64'), fmt='%f', delimiter=',')

        #self.run()
```

Explanation for the above image:

def_init_()

In this function we have defined all the variables to 0. Moreover, we have three joint effort variables ie. self.jnt1_effrt_pub, self.jnt2_effrt_pub, self . jnt3_effrt _ pub which will publish joint torques to the topic

“/rrp/joint1_effort_controller/command”

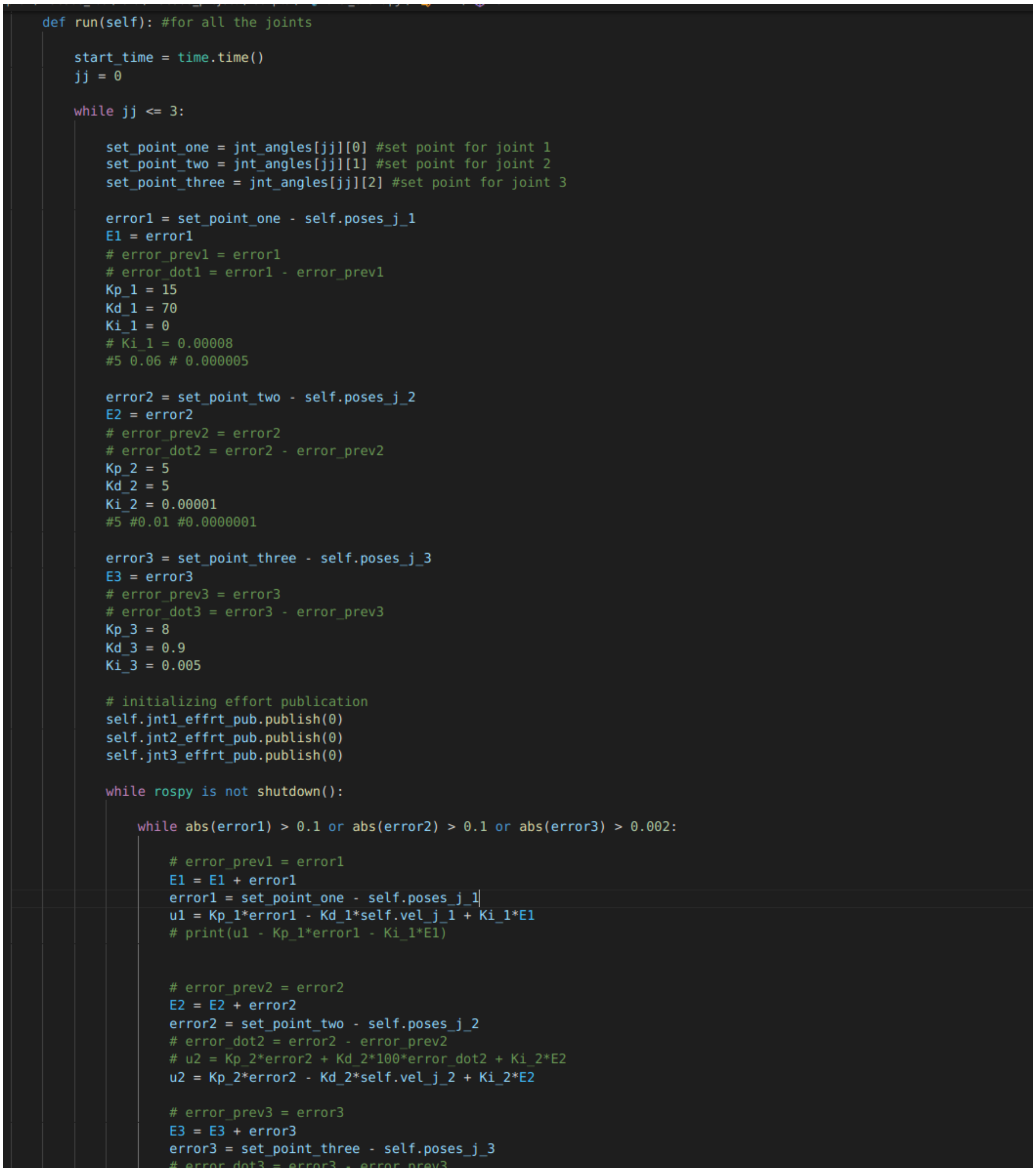
“/rrp/joint2_effort_controller/command”

“/rrp/joint3_effort_controller/command”

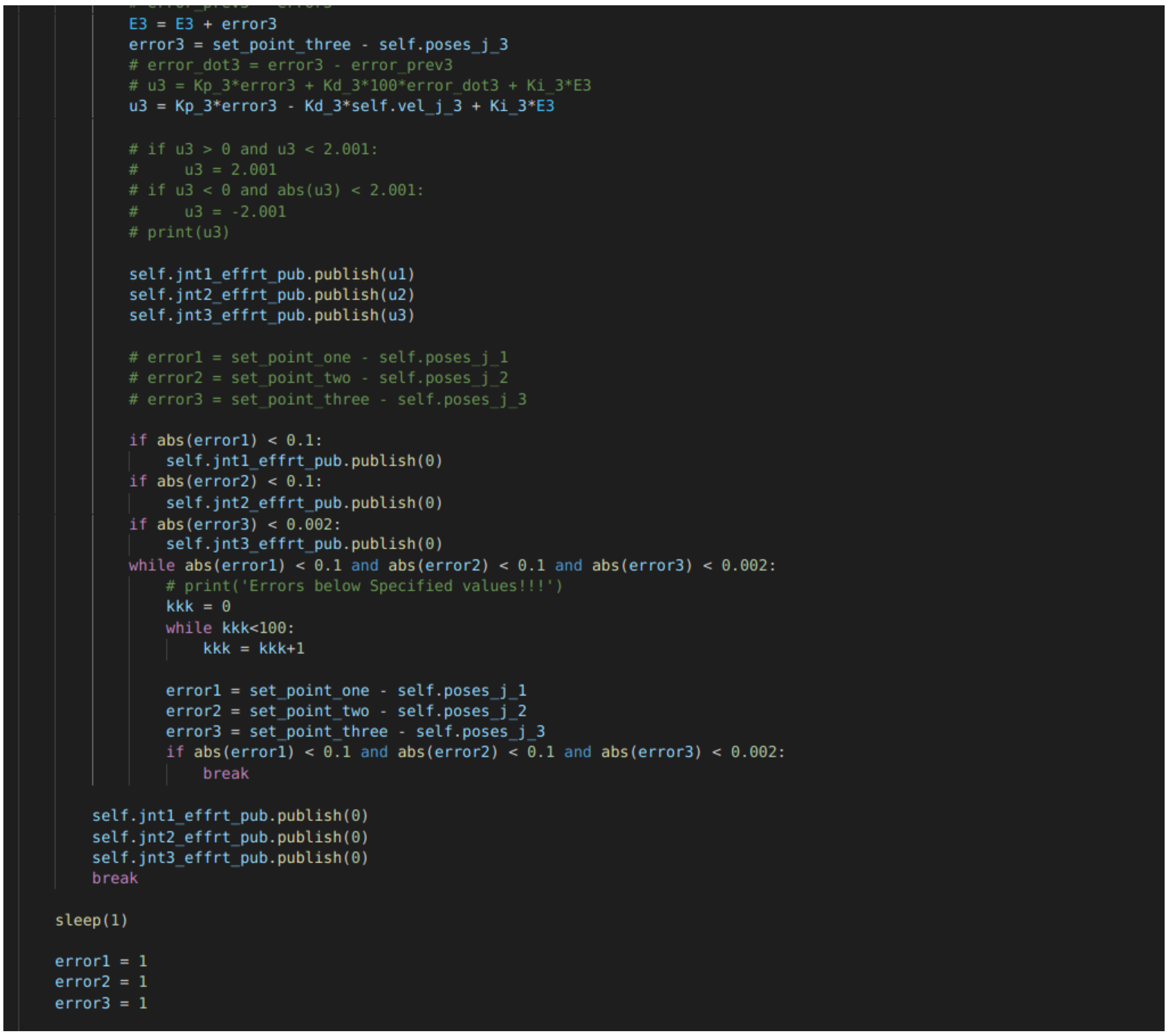
We have set our rospy.rate to 100 as a high control update rate is needed to effectively control the robot joints.

Next, initially, we have defined self.poses_j_1, self.poses_j_2, self.poses_j_3, as 0. Later, these variables will get updated and store the current position of the joints by continuously subscribing to the topic JoinState.

We have also called the run function in this block and will be called whenever class RRP gets called in the main function.



```
    def run(self): #for all the joints

        start_time = time.time()
        jj = 0

        while jj <= 3:

            set_point_one = jnt_angles[jj][0] #set point for joint 1
            set_point_two = jnt_angles[jj][1] #set point for joint 2
            set_point_three = jnt_angles[jj][2] #set point for joint 3

            error1 = set_point_one - self.poses_j_1
            E1 = error1
            # error_prev1 = error1
            # error_dot1 = error1 - error_prev1
            Kp_1 = 15
            Kd_1 = 70
            Ki_1 = 0
            # Ki_1 = 0.00008
            #5 0.06 # 0.000005

            error2 = set_point_two - self.poses_j_2
            E2 = error2
            # error_prev2 = error2
            # error_dot2 = error2 - error_prev2
            Kp_2 = 5
            Kd_2 = 5
            Ki_2 = 0.00001
            #5 #0.01 #0.0000001

            error3 = set_point_three - self.poses_j_3
            E3 = error3
            # error_prev3 = error3
            # error_dot3 = error3 - error_prev3
            Kp_3 = 8
            Kd_3 = 0.9
            Ki_3 = 0.005

            # initializing effort publication
            self.jnt1_effrt_pub.publish(0)
            self.jnt2_effrt_pub.publish(0)
            self.jnt3_effrt_pub.publish(0)

            while rospy is not shutdown():

                while abs(error1) > 0.1 or abs(error2) > 0.1 or abs(error3) > 0.002:

                    # error_prev1 = error1
                    E1 = E1 + error1
                    error1 = set_point_one - self.poses_j_1
                    u1 = Kp_1*error1 - Kd_1*self.vel_j_1 + Ki_1*E1
                    # print(u1 - Kp_1*error1 - Ki_1*E1)


                    # error_prev2 = error2
                    E2 = E2 + error2
                    error2 = set_point_two - self.poses_j_2
                    # error_dot2 = error2 - error_prev2
                    # u2 = Kp_2*error2 + Kd_2*100*error_dot2 + Ki_2*E2
                    u2 = Kp_2*error2 - Kd_2*self.vel_j_2 + Ki_2*E2

                    # error_prev3 = error3
                    E3 = E3 + error3
                    error3 = set_point_three - self.poses_j_3
                    # error_dot3 = error3 - error_prev3
```



```
        E3 = E3 + error3
        error3 = set_point_three - self.poses_j_3
        # error_dot3 = error3 - error_prev3
        # u3 = Kp_3*error3 + Kd_3*100*error_dot3 + Ki_3*E3
        u3 = Kp_3*error3 - Kd_3*self.vel_j_3 + Ki_3*E3

        # if u3 > 0 and u3 < 2.001:
        #     u3 = 2.001
        # if u3 < 0 and abs(u3) < 2.001:
        #     u3 = -2.001
        # print(u3)

        self.jnt1_effrt_pub.publish(u1)
        self.jnt2_effrt_pub.publish(u2)
        self.jnt3_effrt_pub.publish(u3)

        # error1 = set_point_one - self.poses_j_1
        # error2 = set_point_two - self.poses_j_2
        # error3 = set_point_three - self.poses_j_3

        if abs(error1) < 0.1:
            self.jnt1_effrt_pub.publish(0)
        if abs(error2) < 0.1:
            self.jnt2_effrt_pub.publish(0)
        if abs(error3) < 0.002:
            self.jnt3_effrt_pub.publish(0)
        while abs(error1) < 0.1 and abs(error2) < 0.1 and abs(error3) < 0.002:
            # print('Errors below Specified values!!!')
            kkk = 0
            while kkk<100:
                kkk = kkk+1

            error1 = set_point_one - self.poses_j_1
            error2 = set_point_two - self.poses_j_2
            error3 = set_point_three - self.poses_j_3
            if abs(error1) < 0.1 and abs(error2) < 0.1 and abs(error3) < 0.002:
                break

    self.jnt1_effrt_pub.publish(0)
    self.jnt2_effrt_pub.publish(0)
    self.jnt3_effrt_pub.publish(0)
    break

sleep(1)

error1 = 1
error2 = 1
error3 = 1
```

Explanation for the next part of the code:

For the PID Controller part, the logical flow of the PID controller was implemented in the run function. Firstly, a while function was employed to iterate over the four points in the points variable. Then, the tuned P, I and D gains were defined and the error for each of the joints were defined using the js_callback function that returns the joint states as well as the joint velocities.

A separate variable was used to account for the accumulation of error for the I term. Our code controls the values of all three joints simultaneously which can be seen from the condition " while error1>0.1 or error2>0.1 or error3>0.1". The inner while loop was run until the value of the joint errors became less than the pre-specified values; after which, the loop was broken and the next point was taken into consideration. Finally after each iteration, we are resetting the error values so that the values don't get accumulated over time.

The js_callback function also contains the instructions for adding the joint variable values into another variable, trajectory every 100 interactions for plotting.

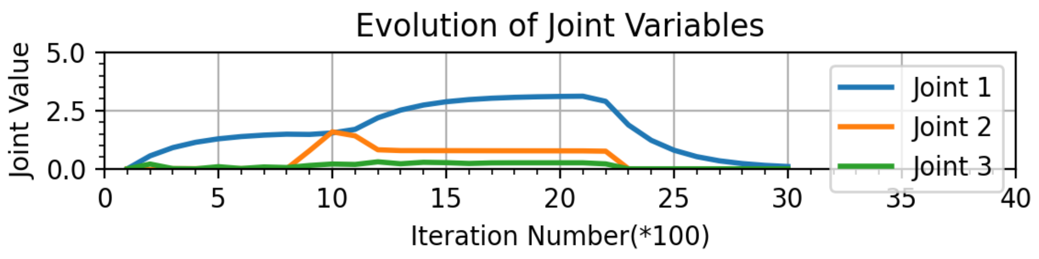
5. Output for Inverse Kinematics

The output for inverse kinematics is given as follows:

End effector positions	**theta_one**	**theta_two**	**d_three**
p1 = [0, 0.77, 0.34]	90.00	0.0	0.049
P2 = [-0.345,0.425,0.24]	89.99	90.0	0.150
P3 = [-0.67, -0.245,0.14]	180.28	44.47	0.25
p4 = [0.77, 0.0, 0.39]	0.0	0.0	0.0

6. Output Trajectory Plots

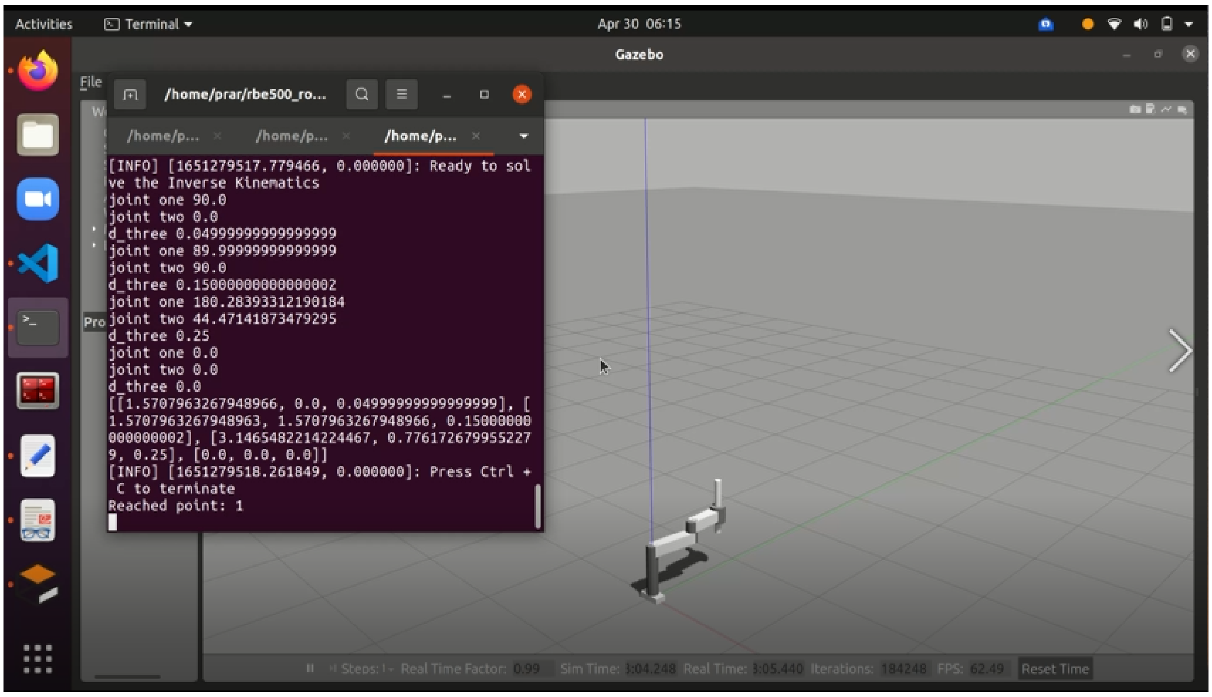
We have plotted the graph of how our joint variables are changing from the beginning to the end. The x-axis shows the number of iterations it takes to reach back to the home position after finishing all the joints. The Y-axis shows all three joint values and the evolution of these values over time. The plotting was done every 100th iteration. Thus, the number of observations plotted is only about 30.



7. Gazebo Simulation Results

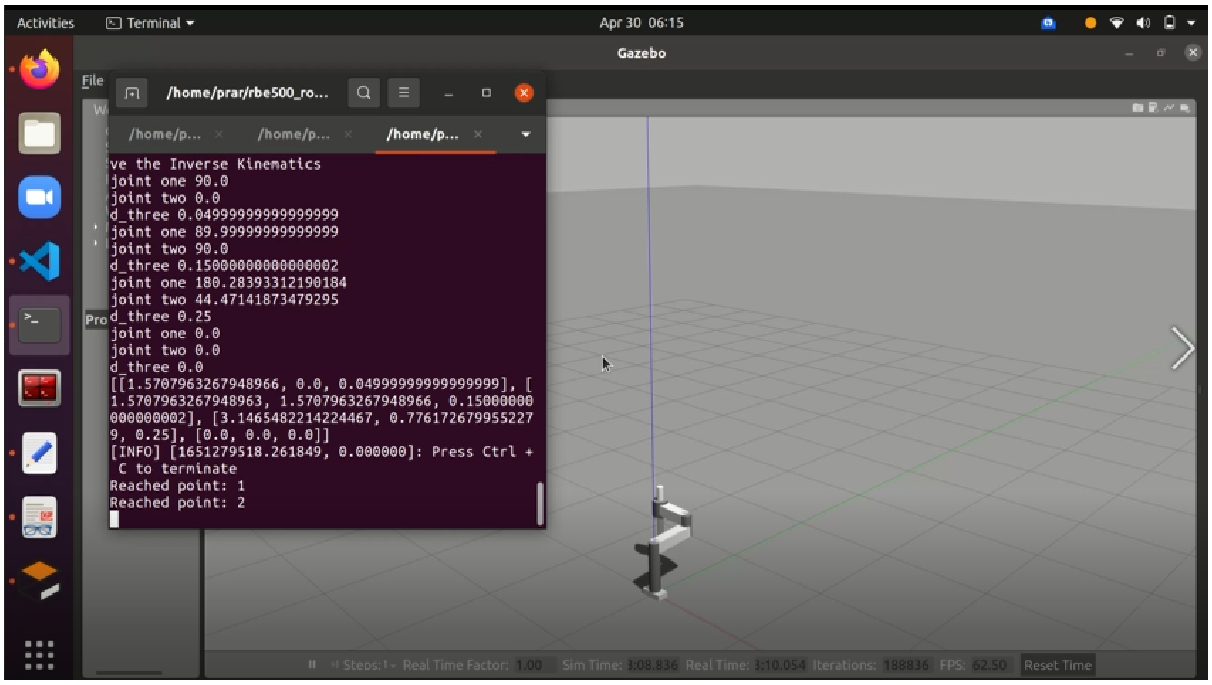
All the four end effector positions have been plotted below:

1) **p1 = [0, 0.77, 0.34]**



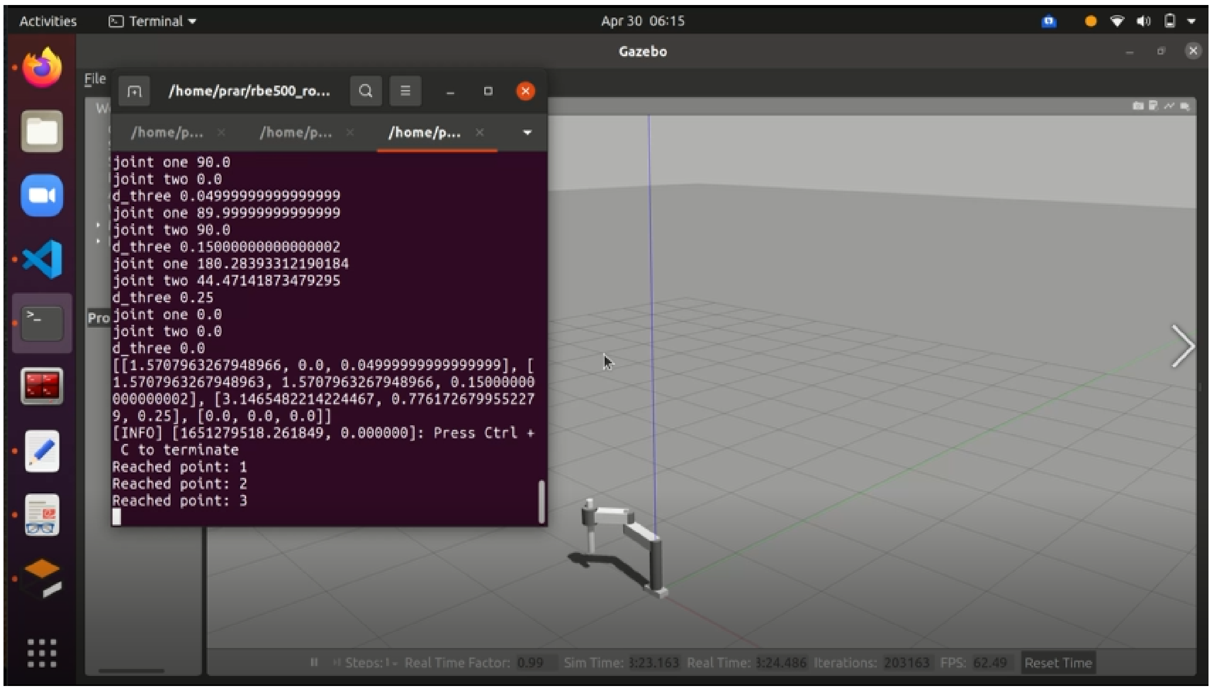
```
[INFO] [1651279517.779466, 0.000000]: Ready to sol
ve the Inverse Kinematics
joint one 90.0
joint two 0.0
d_three 0.04999999999999999
joint one 89.99999999999999
joint two 90.0
d_three 0.15000000000000002
joint one 180.28393312190184
joint two 44.47141873479295
d_three 0.25
joint one 0.0
joint two 0.0
d_three 0.0
[[1.5707963267948966, 0.0, 0.04999999999999999], [
1.5707963267948963, 1.5707963267948966, 0.15000000
000000002], [3.1465482214224467, 0.776172679955227
9, 0.25], [0.0, 0.0, 0.0]]
[INFO] [1651279518.261849, 0.000000]: Press Ctrl +
 C to terminate
Reached point: 1
```

2) **p2 = [-0.345,0.425,0.24]**



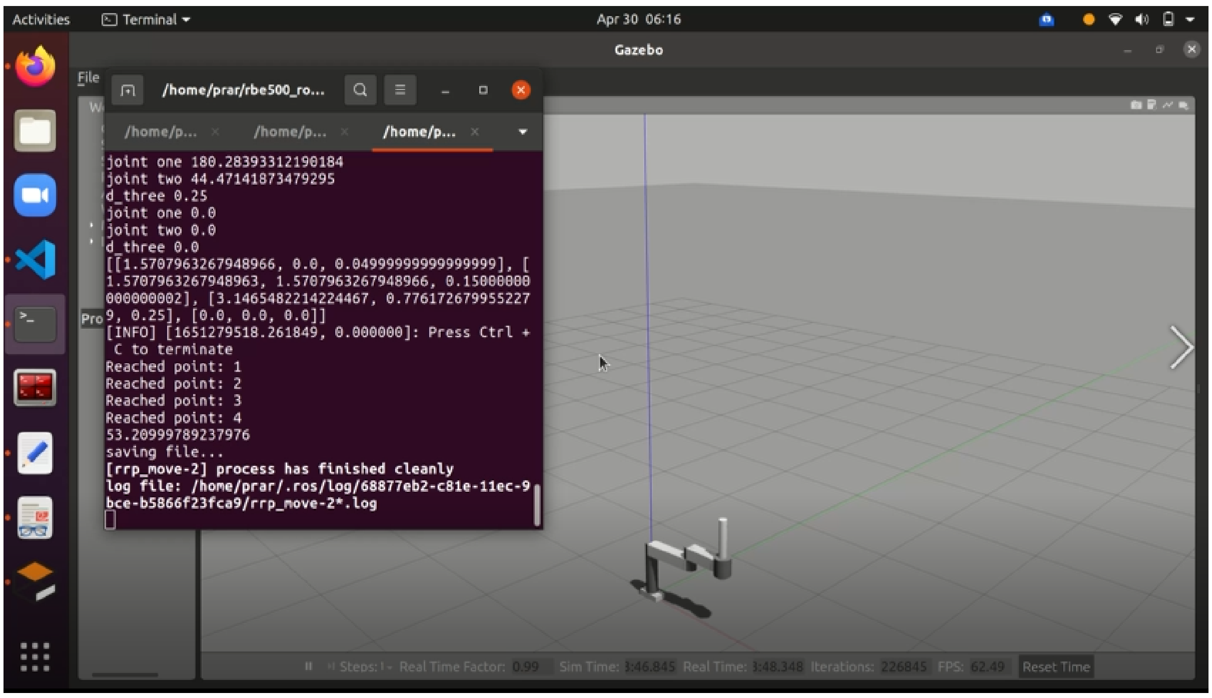
```
ve the Inverse Kinematics
joint one 90.0
joint two 0.0
d_three 0.04999999999999999
joint one 89.99999999999999
joint two 90.0
d_three 0.15000000000000002
joint one 180.28393312190184
joint two 44.47141873479295
d_three 0.25
joint one 0.0
joint two 0.0
d_three 0.0
[[1.5707963267948966, 0.0, 0.04999999999999999], [
1.5707963267948963, 1.5707963267948966, 0.15000000
000000002], [3.1465482214224467, 0.776172679955227
9, 0.25], [0.0, 0.0, 0.0]]
[INFO] [1651279518.261849, 0.000000]: Press Ctrl +
 C to terminate
Reached point: 1
Reached point: 2
```

3) p3 = [-0.67, -0.245,0.14]


Activities
Terminal
Apr 30 06:15
Gazebo
/home/prar/rbe500_ro...
/home/p...
/home/p...
/home/p...
Steps: 1 Real Time Factor: 0.99 Sim Time: 3:23.163 Real Time: 3:24.486 Iterations: 203163 FPS: 62.49 Reset Time

```
joint one 90.0
joint two 0.0
d_three 0.049999999999999999
joint one 89.99999999999999
joint two 90.0
d_three 0.15000000000000002
joint one 180.28393312190184
joint two 44.47141873479295
d_three 0.25
joint one 0.0
joint two 0.0
d_three 0.0
[[1.5707963267948966, 0.0, 0.049999999999999999], [
1.5707963267948963, 1.5707963267948966, 0.15000000
000000002], [3.1465482214224467, 0.776172679955227
9, 0.25], [0.0, 0.0, 0.0]]
[INFO] [1651279518.261849, 0.000000]: Press Ctrl +
 C to terminate
Reached point: 1
Reached point: 2
Reached point: 3
```

4) p4 = [0.77, 0.0, 0.39]


Activities
Terminal
Apr 30 06:16
Gazebo
/home/prar/rbe500_ro...
/home/p...
/home/p...
/home/p...
Steps: 1 Real Time Factor: 0.99 Sim Time: 3:46.845 Real Time: 3:48.348 Iterations: 226845 FPS: 62.49 Reset Time

```
joint one 180.28393312190184
joint two 44.47141873479295
d_three 0.25
joint one 0.0
joint two 0.0
d_three 0.0
[[1.5707963267948966, 0.0, 0.049999999999999999], [
1.5707963267948963, 1.5707963267948966, 0.15000000
000000002], [3.1465482214224467, 0.776172679955227
9, 0.25], [0.0, 0.0, 0.0]]
[INFO] [1651279518.261849, 0.000000]: Press Ctrl +
 C to terminate
Reached point: 1
Reached point: 2
Reached point: 3
Reached point: 4
53.20999789237976
saving file...
[rrp_move-2] process has finished cleanly
log file: /home/prar/.ros/log/68877eb2-c81e-11ec-9
bce-b5866f23fca9/rrp_move-2*.log
```